Corrigé DS5

Exercice

1. Ce problème est indécidable. La preuve vue en cours était attendue.
2. Ce problème est indécidable. En effet, supposons par l'absurde qu'il existe une fonction coarret permettant de résoudre le problème coARRET. Alors la fonction arret suivante permet de résoudre le problème ARRET ce qui est une contradiction vu la question 1 :

```
let arret f x = not (coarret f x)
```

3. Ce problème est indécidable. En effet, supposons par l'absurde qu'il existe une fonction arret_sup10 qui résout ARRET_{10}. Construisons la fonction arret suivante :

```
let arret f x =
    let g (y:int) = f x in
    arret_sup10 g
```

Si le calcul de $f(x)$ termine alors il y a une infinité d'entrées pour lesquelles la fonction g termine et par conséquent arret_sup10 g renvoie true. Si le calcul de $f(x)$ ne termine pas alors g y ne termine pour aucune entrée y et donc arret_sup10 g renvoie false. On en déduit que arret résout correctement le problème de l'arrêt ce qui est une contradiction avec la question 1.

Problème

Partie 1

Cette partie est une partie de l'exercice 6 du TD8 pour lequel j'ai déjà fourni une correction.

Partie 2

4. Le nombre de permutations des sommets est $n!$. Comme un cycle hamiltonien commence à partir de n'importe quel sommet et peut être parcouru dans les deux sens (le graphe étant non orienté), à chaque cycle correspondent $2n$ permutations. On en déduit que le nombre de cycles hamiltoniens différents est $(n-1)!/2$.
5. Il s'agit de récupérer le coefficient qui se trouve en position (s,t) dans la matrice d'adjacence ; ce dernier se trouve en case $s \times n + t$ dans le tableau correspondant au graphe :

```
int f(int* G, int n, int s, int t)
{
    return G[s*n+t];
}
```

6. On somme les poids (calculés grâce à la fonction précédente) de toutes les arêtes intervenant dans le cycle. Attention à ne pas oublier l'arête fermant le cycle (entre le dernier sommet et le premier).

```
int poids_cycle(int* G, int* c, int n)
{
    int poids = f(G,n,c[0],c[n-1]);
    for (int i = 0; i < n-1; i++)
    {
        poids = poids + f(G,n,c[i],c[i+1]);
```

```
    }
    return poids;
}
```

7. On propose l'algorithme suivant :
   - Calculer j : on initialise j à $n-2$ et on le décrémente tant qu'il est positif et que $p_j > p_{j+1}$.
   - Si $j = -1$, les éléments formant la permutation actuelle sont triés dans l'ordre décroissant : on est donc face à la dernière permutation selon l'ordre lexicographique et on renvoie alors faux.
   - Sinon :
     - Calculer k : on initialise k à $n-1$ et on le décrémente tant que $p_j > p_k$.
     - Echanger les valeurs de p_j et p_k.
     - Renverser les valeurs comprises entre les indices $j+1$ et $n-1$.
     - Renvoyer vrai.

8. On commence par créer deux tableaux : l'un pour stocker la premutation courante (initialisé avec la première permutation selon l'ordre lexicographique), l'autre pour stocker la permutation correspondant au meilleur cycle hamiltonien trouvé pour le moment. Tant qu'on n'a pas atteint la dernière permutation, on calcule le poids du cycle correspondant à la permutation courante et on met à jour poids_min et cmin le cas échéant. On libère le tableau non renvoyé avant la fin.

```
int* PVC_naif(int* G, int n)
{
    int* c = (int*)malloc(sizeof(int)*n);
    int* cmin = (int*)malloc(sizeof(int)*n);
    for (int i = 0; i < n; i++)
    {
        c[i] = i;
        cmin[i] = i;
    }
    int poids_min = poids_cycle(G,c,n);
    while (permutation_suivante(c,n))
    {
        int poids = poids_cycle(G,c,n);
        if (poids < poids_min)
        {
            poids_min = poids;
            for (int i = 0; i < n; i++)
            {
                cmin[i] = c[i];
            }
        }
    }
    free(c);
    return cmin;
}
```

9. Notons n le nombre de sommets du graphe en entrée. La boucle tant que de PVC_naif est exécutée autant de fois qu'il y a de permutation de $[\![0, n-1]\!]$, soit $n!$ fois. A chaque itération, on fait un

appel à permutation_suivante en $O(1)$, un appel à poids_cycle en $O(n)$ et une éventuelle recopie de tableau en $O(n)$. La complexité totale est donc en $O(n \times n!)$.

Partie 3

10. En partant de 0, on obtient le cycle $(0, 1, 2, 4, 3)$, qui est de poids 23. En partant de 3, on obtient le cycle $(3, 2, 4, 0, 1)$, qui est de poids 22. Aucun de ces cycles n'est de poids minimal puisque le cycle $(0, 1, 4, 2, 3)$ est de poids strictement inférieur (égal à 18).

11. On commence par trouver un sommet non vu, et il en existe un par hypothèse ce qui garantit la terminaison de la boucle tant que ci-dessous. Puis, on parcourt les sommets restants en gardant en mémoire le sommet non vu qui minimise le poids de l'arête menant à s. On considère que le sommet s est vu : c'est cohérent avec l'utilisation qu'on fera de plus_proche dans les fonctions suivantes.

```
int plus_proche(int* G, bool* vus, int n, int s)
{
    int tmin = 0;
    while (vus[tmin]) {tmin++;}
    for (int t = tmin; t < n; t++)
    {
        if (!vus[t] && f(G,n,s,t) < f(G,n,s,tmin)) {tmin = t;}
    }
    return tmin;
}
```

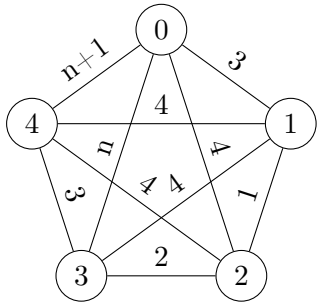
12. On créé un tableau qui contiendra le cycle final, et un pour garder en mémoire les sommets déjà intégrés au cycle. Il ne reste plus qu'à parcourir les sommets à partir de 0 (c[0] contiendra bien 0 vu l'initialisation de c) selon l'heuristique du plus proche voisin.

```
int* PVC_glouton(int* G, int n)
{
    int* c = (int*)malloc(sizeof(int)*n);
    bool* vus = (bool*)malloc(sizeof(bool)*n);
    for (int i = 0; i < n; i++)
    {
        c[i] = 0;
        vus[i] = false;
    }
    vus[0] = true;
    for (int i = 1; i < n; i++)
    {
        c[i] = plus_proche(G, vus, n, c[i-1]);
        vus[c[i]] = true;
    }
    free(vus);
    return c;
}
```

13. Notons n le nombre de sommets du graphe en entrée de PVC_glouton. L'initialisation de c et vus se fait en temps $O(n)$ puis on fait $n-1$ appels à plus_proche, dont la complexité est clairement en $O(n)$. On obtient donc une complexité pour PVC_glouton en $O(n^2)$.

Remarque : La complexité de cet algorithme est donc linéaire en la taille du graphe en entrée puisque ce dernier a n^2 arêtes étant donné qu'il est complet.

14. Pour tout $n \geq 5$, on considère l'instance G_n de PVC définie par le graphe suivant :



L'heuristique du plus proche voisin partant de 0 renvoie le cycle $(0,1,2,3,4)$ de poids $P_n = n+10$. Mais dans G_n, le cycle $(0,1,4,3,2)$ a pour poids 14 ce qui garantit que le poids minimal $P_n^\star$ d'un cycle hamiltonien dans G_n est inférieur à 14.

S'il existait $\alpha \geq 1$ tel que PVC_glouton soit une α-approximation pour PVC, on aurait pour tout $n \geq 5$, $P_n \leq \alpha P_n^\star$. Mais c'est impossible puisque $P_n/P_n^\star \to +\infty$ lorsque $n \to +\infty$ d'après ce qui précède. On en déduit que PVC_glouton n'est pas une approximation de PVC.

Remarque : C'est cohérent avec le résultat de la partie 7 !

15. Avec les notations introduites à la question 14, le graphe G_{100} fournit un exemple convenable :

Sommet de départ	Cycle C obtenu par PVC_glouton	Poids de C
0	$(0,1,2,3,4)$	110
1	$(1,2,3,4,0)$	110
2	$(2,1,0,3,4)$	111
3	$(3,2,1,0,4)$	110
4	$(4,3,2,1,0)$	110

Comme le cycle $(0,1,4,3,2)$ est de poids 14, il est certain qu'aucun des cycles précédents n'est de poids minimal et donc l'heuristique du plus proche voisin échoue à trouver un cycle hamiltonien de poids minimal pour chacun des sommets de départ.

Remarque : On vient d'ailleurs de montrer qu'il existe des instances pour lesquelles, quel que soit le sommet de départ, le poids du cycle calculé via l'heuristique du plus proche voisin peut être aussi éloigné du poids optimal que souhaité. On pouvait espérer qu'au moins l'un des n cycles calculés via cette heuristique ait un poids faible : il n'en est en fait rien.

Partie 4

16. Les chemins de 0 à s passant une et une seule fois par chaque sommet de S' se partitionnent en

$$\bigsqcup_{s' \in S' \text{ voisin de } s} \{\text{chemins allant de 0 à } s' \text{ passant une et une seule fois par chaque sommet de } S' \setminus \{s'\}\}$$

puisque sur un tel chemin, s a forcément un prédecesseur dans S'. On a donc :

$$P_{\min}(s, S') = \min_{s' \in S'} \big(P_{\min}(s', S' \setminus \{s'\}) + f(s, s')\big)$$

et tout sommet s' pour lequel ce minimum est atteint est égal à $\text{pred}(s, S')$.

17. Si cle est un objet mutable, qu'on créé une association entre cle et une valeur v puis qu'on modifie (potentiellement par mégarde) cle, la valeur v ne peut plus être retrouvée avec la variable cle comme clé. Une formulation plus imagée : si vous vous faitez livrer un colis à votre adresse puis que vous changez d'adresse, vous ne retrouverez pas votre colis.

18. Un tableau de booléens de taille n peut être assimilé à la décomposition en base deux d'un entier entre 0 et $2^n - 1$. On exploite cette correspondance pour encoder de manière unique chaque sous ensemble S'. Pour encoder de manière unique un couple (s, S'), on utilise tout simplement (s, encodage de S').

```
let encodage (s:int) (ensemble:bool array) :int*int =
  let code_ensemble = ref 0 in
  for i = 0 to Array.length ensemble -1 do
    code_ensemble := 2* !code_ensemble;
    if ensemble.(i) then code_ensemble := !code_ensemble +1;
  done;
  (s, !code_ensemble)
```

19. Le principe est le suivant : on calcule la clé k correspondant au couple (s, S') en entrée à l'aide de encodage. Si cette clé n'est pas présente dans tabh, on calcule la valeur qui lui est associée et on la stocke dans tabh. Une fois cette opération faite, il existe une association ayant pour clé k dans tabh (soit parce que c'était déjà le cas, soit parce qu'on vient de la calculer) : on renvoie la valeur qui est associée à k.

```
let rec chemin_min (sommet:int) (ensemble:bool array) :int*int =
  let (s,code) = encodage sommet ensemble in
  if not (Hashtbl.mem tabh (s,code)) then
    begin
      if code = 0 then Hashtbl.add tabh (s,code) (g.(0).(sommet),0)
      else
        begin
          let poids_min = ref max_int in
          let smin = ref 1 in
          for s' = 1 to Array.length ensemble -1 do
            if ensemble.(s') then
              begin
                ensemble.(s') <- false;
                let poids = fst (chemin_min s' ensemble) + g.(s').(s) in
                ensemble.(s') <- true;
                if poids < !poids_min then
                  (poids_min := poids; smin := s')
              end
          done;
          Hashtbl.add tabh (s,code) (!poids_min, !smin);
        end
    end;
  Hashtbl.find tabh (s,code)
```

Le calcul d'une valeur à associer à une clé donnée se fait comme suit :

- Le cas de base se produit quand $S' = \emptyset$ (ce qu'on teste en vérifiant que le code associé à S' vaut 0) : dans ces conditions, $P_{\min}(s, S')$ est le poids d'un plus court chemin allant de 0 à s

sans passer par aucun autre sommet : il s'agit donc de poids de l'arête $(0, s)$ et le prédecesseur de s est 0. Ceci est valable y compris lorsque s vaut 0.

- Le cas récursif se gère en s'aidant de la question 16. Pour chaque sommet s' dans S', on calcule $P_{\min}(s', S' \setminus \{s'\})$ (pour calculer $S' \setminus \{s'\}$, on retire temporairement s' à ensemble et on le rajoute une fois le calcul du $P_{\min}$ souhaité effectué) puis on met à jour le poids minimal trouvé et le sommet par lequel ce poids minimal est atteint le cas échéant.

20. On commence par créer un tableau de booléens ensemble encodant l'ensemble $S' = S \setminus \{0\}$ et un tableau cmin destiné à contenir le cycle souhaité. On reconstruit un chemin de 0 à 0 passant une et une seule fois par les sommets de S' en commençant par la fin : pour ce faire, on récupère successivement les prédecesseurs de 0 sur un chemin minimal grâce à chemin_min.

```
let pvc_dynamique () :int array =
  let n = Array.length g in
  let ensemble = Array.init n (fun i -> i <> 0) in
  let cmin = Array.make n 0 in
  let pred = ref (snd (chemin_min 0 ensemble)) in
  for i = n-1 downto 1 do
    cmin.(i) <- !pred;
    ensemble.(!pred) <- false;
    pred := snd (chemin_min !pred ensemble)
  done;
  cmin
```

21. Notons n le nombre de sommets du graphe considéré par pvc_dynamique.

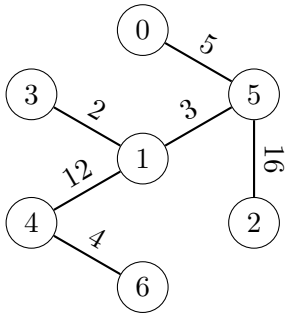
Pour tout $s \in [\![0, n-1]\!]$ et tout $S' \subset S \setminus \{0\}$, on devra calculer $P_{\min}(s, S')$. Or, il y a $n2^{n-1}$ tels couples (s, S') et, pour chacun, le premier calcul de $(P_{\min}(s, S'), \text{pred}(s, S'))$ nécessite un parcours d'un tableau de taille n et au plus n appels à chemin_min, lesquels se font en $O(1)$ grâce à la mémoïsation. Le calcul de tous les $(P_{\min}(s, S'), \text{pred}(s, S'))$ nécessaires s'effectue donc en temps $O(n^2 2^n)$. Une fois ces valeurs calculées, la reconstruction d'un cycle hamiltonien de poids minimal se fait linéairement en n. Donc la complexité temporelle de pvc_dynamique est en $O(n^2 2^n)$.

La complexité spatiale de cet algorithme correspond au nombre de valeurs stockées lors de la mémoïsation donc au nombre de couples $(s, S') \in [\![0, n-1]\!] \times S \setminus \{0\}$: elle est en $O(n2^n)$.

Remarque : La complexité temporelle de l'algorithme de Held-Karp est bien meilleure que celle établie à la question 9 mais reste exponentielle ; et c'est normal. On observe au passage une illustration du compromis temps-mémoire.

Partie 5

22. On obtient l'arbre couvrant suivant :



23. On peut implémenter l'algorithme de Prim comme suit :

    - L'arbre renvoyé sera représenté par un graphe lui même représenté par listes d'adjacence : pour ajouter l'arête (s,t) dans l'arbre, il suffira donc d'ajouter s dans la liste d'adjacence de t et t dans la liste d'adjacence de s.
    - L'ensemble R est représenté par un tableau de booléens : la case s de R contient `true` si et seulement si le sommet s appartient à R.
    - On utilise enfin une file de priorité min F pour faciliter le calcul de l'arête (s,t) lors de chaque itération de la boucle tant que : elle contient des arêtes et la priorité d'une arête est son poids. On initialise F avec les arêtes incidentes à s_0. La boucle tant que consiste à extraire l'arête (s,t) de priorité minimale de F : on vérifie si $s \in R$ et $t \notin R$ (ou inversement).
        - Si c'est le cas, on ajoute t à R, on ajoute (s,t) à l'arbre en construction et on ajoute toutes les arêtes incidentes à t dans F.
        - Sinon, on ne fait rien et on passe à l'itération suivante.

    Dans ces conditions, chaque arête de G passera dans la file de priorité au plus une fois par extrémité de l'arête, c'est-à-dire deux fois. Si on implémente cette file via un tas min, la complexité de l'algorithme de Prim s'estime comme suit :

    - La création du tableau de listes d'adjacence qui encodera l'arbre renvoyé et la création de R se font en $O(|S|)$.
    - La boucle tant que consiste à faire au plus $2|A|$ insertions et extractions dans une file de priorité min de taille majorée par $2|A|$ et au plus $|S|-1$ ajouts d'arêtes à l'arbre en construction, chacun en temps constant. Cette boucle exige donc de faire de l'ordre de $|S|+2|A|\log(2|A|)$ opérations.

    Finalement, le graphe G étant connexe, $|S| \leq |A|+1$ donc la complexité totale de l'algorithme de Prim est en $O(|A|\log|S|)$ (on a en effet $\log|A| \leq 2\log|S|$ systématiquement).

24. La propriété "(S,B) est connexe" est un invariant immédiat de boucle pour la boucle tant que. De plus, à chaque itération de cette boucle, on ajoute un nouveau sommet à R ce qui implique que cette boucle s'exéccute $|S|-1$ fois donc qu'en fin d'algorithme $|B| = |S|-1$.

    Ainsi, en fin d'algorithme, (S,B) est connexe et $|B| = |S|-1$: c'est donc un arbre. Comme son ensemble de sommets est S, c'est par définition un arbre couvrant G.

25. Notons $a_i = (s,t)$. On peut loisiblement supposer que $s \in R$ et $t \notin R$. Comme $T^\star$ est un arbre, il est connexe donc il existe un chemin dans $T^\star$ entre s et t. Comme $s \in R$ et $t \notin R$, il existe une arête a sur ce chemin reliant un sommet de R à un sommet qui n'est pas dans R. On a nécessairement $f(a) \geq f(a_i)$ sinon l'algorithme de Prim aurait choisi d'ajouter l'arête a à l'arbre en construction plutôt que a_i.

    Considérons le graphe $T' = (S, B^\star \cup \{a_i\} \setminus \{a\})$. Alors :

    - T' est connexe.
    - T' possède $|S|-1$ arêtes puisque $a_i \notin B^\star$ et $a \in B^\star$. Combiné avec le premier point, cela montre que T' est un arbre couvrant de G.
    - $f(T') = f(T^\star) + f(a_i) - f(a) \leq f(T^\star)$ de par l'inégalité existant entre le poids de a et celui de a_i. Comme $T^\star$ est minimal, T' aussi.

    Ainsi, T' est un arbre couvrant minimal de G et la première arête choisie par l'algorithme de Prim qui n'apparaît pas dans T' a un indice strictement plus grand que i. Ceci est contradictoire avec

la définition de $T^\star$ et montre que notre hypothèse initiale était absurde : T est un arbre couvrant minimal de G.

26. Les détails d'implémentation ont déjà été fournis en question 23. On choisit arbitrairement de commencer la construction de l'arbre à partir du sommet 0. A la place d'itérer jusqu'à ce que la file de priorité soit vide, on itère jusqu'à ce qu'on ait ajouté $n-1$ arêtes à notre arbre couvrant : on sait que dès que cela arrive, tous les sommets sont dans R et donc que les arêtes qui restent dans la file de priorité à ce moment ne serait de toutes façons que défilées sans avoir d'impact sur le résultat.

```
let prim (g:graphe) :graphe =
  let n = Array.length g in
  let acm = Array.make n [] in
  let fp = creer_FP () in
  let r = Array.make n false in
  r.(0) <- true;
  let ajout_arete s (t,p) = ajouter_FP (s,t) p in
  List.iter (ajout_arete 0) g.(0);
  let nb_ajouts = ref 0 in
  while !nb_ajouts < n - 1 do
    let (s,t) = extraire_FP fp in
    if not r.(t) then
      begin
        r.(t) <- true;
        acm.(s) <- t::acm.(s);
        asm.(t) <- s::acm.(t);
        nb_ajouts := !nb_ajouts +1;
        List.iter (ajout_arete t) g.(t)
      end
  done;
  acm
```

27. Soit $s,t \in S$. Par l'inégalité triangulaire, $f(s,t) \leq f(s,t) + f(t,t)$ donc $f(t,t) \geq 0$. Toujours par l'inégalité triangulaire, on a donc :

$$0 \leq f(t,t) \leq f(t,s) + f(s,t) = 2f(s,t) \quad \text{puisque le graphe est non orienté,}$$

donc $f(s,t) \geq 0$.

28. Comme $c^\star$ est un cycle hamiltonien, si on supprime une arête a de $c^\star$, on obtient un arbre couvrant T' de G. Comme T est un arbre couvrant minimal, on a donc :

$$f(T) \leq f(T') = f(c^\star) - f(a) \leq f(c^\star)$$

la dernière inégalité provenant de ce que $f(a) \geq 0$ d'après la question 27.

29. On propose l'algorithme suivant :

    - Calculer un arbre couvrant minimal T de G (peut se faire en $O(|A| \log |S|)$ opérations avec l'algorithme de Prim d'après la question 23).
    - Calculer un parcours préfixe des sommets de T (peut se faire en $O(|S|)$ opérations) et renvoyer les sommets dans cet ordre. Ceci produit un cycle c_T hamiltonien puisque G est complet.

    Cet algorithme est effectivement polynomial en la taille de G.

    Par ailleurs, $f(c_T) \leq 2 \times f(T)$. D'après la question 28, on a ainsi $f(c_T) \leq 2f(c^\star)$ ce qui conclut.

Exo : Montrer rigoureusement que $f(c_T) \leq 2f(T)$. Le point critique est l'utilisation de l'inégalité triangulaire. Constater que l'intuition donnée via le dessin en cours est bien plus parlante !

30. On suit le principe exposé dans la question 29 : on calcule un arbre couvrant minimal à l'aide de prim puis on effectue un parcours en profondeur sur cet arbre en stockant les sommets dans cT dans l'ordre dans lequel ils sont découverts.

```
let pvc_approx (g:graphe) =
  let n = Array.length g in
  let vus = Array.make n false in
  let cT = Array.make n 0 in
  let acm = prim g in
  let i = ref 0 in
  let rec parcours_pronfondeur (s:int) =
    if not vus.(s) then begin
        vus.(s) <- true;
        cT[!i] = s;
        i := !i +1;
        List.iter parcours_profondeur acm.(s)
      end
  in
  parcours_profondeur 0;
  cT
```

La question 29 assure que cette fonction renvoie une 2-approximation du cycle hamiltonien optimal en temps polynomial en la taille du graphe en entrée.

Partie 7

31. On procède par double implication. Si G possède un cycle hamiltonien alors ce cycle est toujours un cycle hamiltonien dans K_G et il emprunte $|S|$ arêtes (puisque $|S| > 2$) qui sont toutes de poids 1 donc son poids est $|S| \leq \alpha|S|$ puisque $\alpha > 1$.

Réciproquement, supposons que K_G admet un cycle hamiltonien de poids inférieur ou égal à $\alpha|S|$. Ce cycle implique au moins deux arêtes (puisque $|S| > 2$). S'il empruntait au moins une arête de poids $\alpha|S|$, alors son poids serait au moins égal à $\alpha|S|+1$ ce qui est une contradiction. On en déduit que ce cycle n'emprunte que des arêtes de poids 1 donc que ce cycle existe dans G.

32. Voici dans ces conditions un algorithme polynomial résolvant CH. Si G est une instance de CH :

- Calculer K_G. Ceci peut se faire en temps $O(|S|^2)$ en parcourant S^2 et en vérifiant pour chaque couple (s,t) s'il correspond à une arête de G (ce qui peut se faire en temps constant si G est représenté par matrice d'adjacence).
- Appliquer l'α-approximation de PVC à K_G : elle calcule un cycle hamiltonien c dans K_G en temps polynomial en $|K_G|$ donc en $|G|$ par hypothèse et tel que $f(c) \leq \alpha f(c^\star)$.
- Si $f(c) \leq \alpha|S|$ alors G admet un cycle hamiltonien d'après la question précédente : on renvoie vrai. Sinon, $\alpha|S| < f(c) \leq \alpha f(c^\star)$ donc $|S| < f(c^\star)$ et par conséquent il n'existe pas dans K_G de cycle hamiltonien qui ne passe que par des arêtes de poids 1, donc pas de cycle hamiltonien dans G non plus : on renvoit faux.

Cet algorithme est bien polynomial en $|G|$ et résout CH. Si P $\neq$ NP, ceci est une contradiction car l'appartenance à P d'un problème NP-complet montre que ces deux classes sont égales. Si P $\neq$ NP, on vient donc de montrer qu'il n'existe aucune approximation à facteur constant pour PVC.